Introduction to PIC Programming

Midrange Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 3: Reading Switches

The first lesson introduced simple digital output, by turning on or flashing an LED. That's more useful than you may think, since, with some circuit changes (such as adding transistors and relays), it can be readily adapted to turning on and off almost any electrical device.

Most systems, however, need to interact with their environment in some way; to respond to user commands or varying inputs. The simplest form of input is an on/off switch. This lesson revisits the material covered in baseline lesson 4, showing how to read a simple pushbutton switch – techniques which are applicable to any digital (strictly on/off or high/low) input.

This lesson covers:

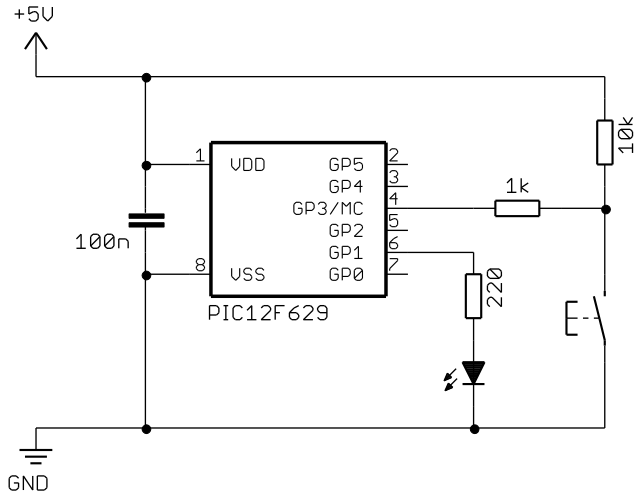
- Reading digital inputs
- Conditional branching
- Using internal pull-ups
- Software approaches to switch debouncing

Example 1: Reading a Digital Input

One of the simplest ways to generate a digital input is to use a basic pushbutton switch.

To demonstrate how to read and respond to a pushbutton switch, we can add one to the "LED flashing" circuit used in lesson 1.

Fortunately the Low Pin Count demo board used for these lessons already includes a tact switch connected to pin GP3, as shown below. You should keep the LED from the previous lessons connected to GP1.



The pushbutton is connected to pin 4 (GP3) via a 1 kΩ resistor. As explained in baseline lesson 4, this provides some protection against electro-static discharge (*ESD*, which pushbuttons, among other devices, can be susceptible to). Resistors like this are also used to avoid damage in case an input pin is inadvertently programmed as an output. Such damage is impossible in this case because, as mentioned in lesson 1, GP3 can only ever be an input. The most important reason for the resistor between pin 4 and the pushbutton is to allow the PIC to be safely and successfully programmed by the PICkit 2, using the ICSP programming

protocol, when pin 4 is used as the 'VPP' input. During ICSP programming, a high voltage (around 12 V) is applied to VPP, to place the PIC into programming mode. The 1 kΩ resistor is necessary to protect the PICkit 2, in case the pushbutton on the LPC Demo Board is pressed during programming, grounding the VPP (12 V) signal.

Pins 6 and 7 are also used in ICSP programming (ICSPCLK and ICSPDAT, respectively); the PICkit 2 manual provides details of the type of isolation circuitry required on these lines, but typically a simple resistor is sufficient.

This is not a consideration on the remaining pins. If you know what you are doing and understand the risk from ESD, you can leave out protection resistors on switch inputs, such as the 1 kΩ resistor on GP3.

The 10 kΩ resistor is a *pull-up* resistor, holding GP3 high while the switch is open.

When the switch is pressed, the pin is pulled to ground through the 1 kΩ resistor.

Given the high impedance of the PIC's inputs (very little current flows into them), these external resistors are sufficient to pull the input voltage to a valid logic high when the pushbutton is up, and a valid logic low when it is pressed. For a more detailed analysis, see baseline lesson 4.

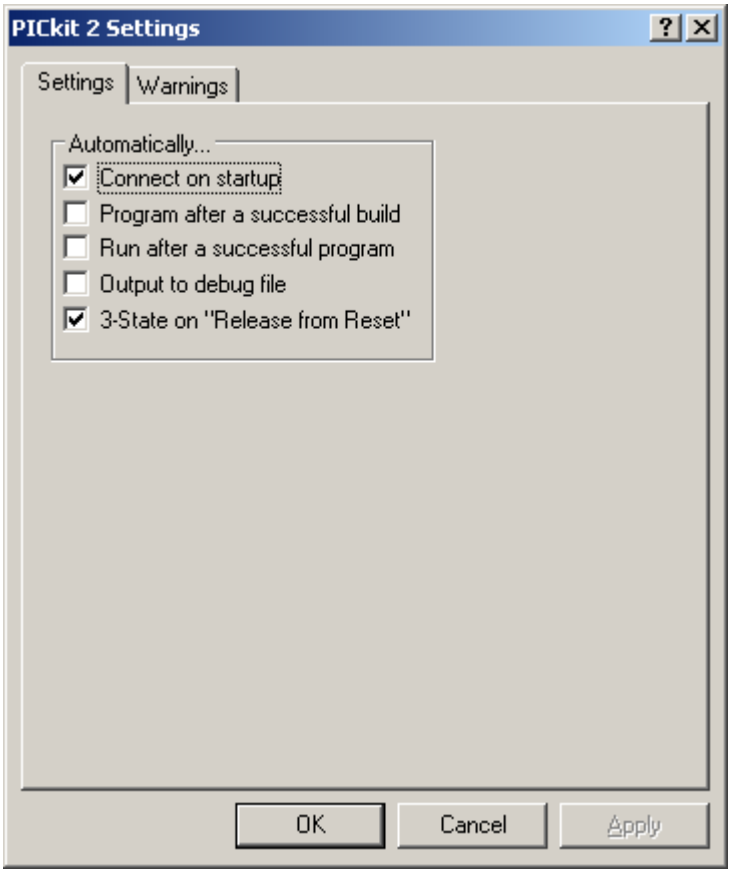
Interference from $\overline{MCLR}$

There is a potential problem with using a pushbutton on GP3; as we have seen, the same pin can instead be configured (using the PIC's configuration word) as the processor reset line, $\overline{MCLR}$.

When the PICkit 2 is used as a programmer from within MPLAB, it holds the $\overline{MCLR}$ line low after programming, until you select "Release from Reset", which, by default, makes the $\overline{MCLR}$ line go high. Either way, the PICkit 2 is asserting control over the $\overline{MCLR}$ line, connected directly to pin 4 (GP3), and, because of the 1 kΩ isolation resistor, the 10 kΩ pull-up resistor and the pushbutton cannot overcome the PICkit 2's control of that line.

If you are using MPLAB 8.10 or later, this problem can be overcome by changing the PICkit 2 programming settings, to tri-state the PICkit 2's $\overline{MCLR}$ output (effectively disconnecting it) when it is not being used to hold the PIC in reset.

To do this, select the PICkit 2 as a programmer (using the "Programmer → Select Programmer" submenu) and then use the "Programmer → Settings" menu item to display the PICkit 2 Settings dialog window, shown on the right.

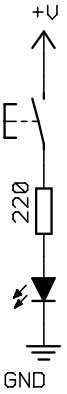


Select the '3-State on "Release from Reset"' option in the 'Settings' tab and then click on the 'OK' button.

When you now click on the on the  icon in the programming toolbar, or select the "Programmer → Release from Reset" menu item, the PICkit 2 will release control of the reset line, allowing GP3 to be driven high or low by the pull-up resistor and pushbutton.

Reading the Switch

We'll start with a short program that simply turns the LED on when the pushbutton is pressed.



Of course, that's a waste of a microcontroller. To get the same effect, you could leave the PIC out and build the circuit shown on the right! But, this simple example avoids having to deal with the problem of switch contact bounce, which we'll look at later.

In general, to read a pin, we need to:

- Configure the pin as an input
- Read or test the bit corresponding to the pin

Recall, from lesson 1, that the pins on the 12F629 are *digital* inputs or outputs. They can be turned on or off, but nothing in between. Similarly, they can read only a voltage as being "high" or "low". The data sheet defines input voltage ranges where the pin is guaranteed to read as "high" or "low". For voltages between these ranges, the pin might read as either; the input behaviour for intermediate voltages is *undefined*.

As you might expect, a "high" input voltage reads as a '1', and a "low" reads as a '0'.

Normally, to configure a pin as an input, you would set the corresponding TRISIO bit to '1'. However, this circuit uses GP3, which, because it shares a pin with $\overline{\text{MCLR}}$, can only ever be an input – regardless of the contents of TRISIO. However, when using GP3 as an input, you may as well set bit 3 of TRISIO, to make your code clearer.

An instruction such as '`movf GPIO,w`' will read the bit corresponding to GP3. The problem with that is that it reads all the pins in GPIO, not just GP3. If you want to act only on a single bit, you need to separate it from the rest, which can be done with logical masking and shift instructions, but there's a much easier way – use the bit test instructions. There are two:

'`btfsc f,b`' tests bit 'b' in register 'f'. If it is '0', the following instruction is skipped – "**b**it **t**est **f**ile register, **s**kip if **c**lear".

'`btfss f,b`' tests bit 'b' in register 'f'. If it is '1', the following instruction is skipped – "**b**it **t**est **f**ile register, **s**kip if **s**et".

Their use is illustrated in the following code:

```
        ; initialisation
        movlw   ~(1<<GP1)       ; configure GP1 (only) as an output
        banksel TRISIO          ; (GP3 is an input)
        movwf   TRISIO
        banksel GPIO
        clrf    GPIO            ; start with GPIO clear (GP1 low)

;***** Main loop
loop
        btfss   GPIO,GP3        ; if button pressed (GP3 low)
        bsf     GPIO,GP1        ;   turn on LED
        btfsc   GPIO,GP3        ; if button up (GP3 high)
        bcf     GPIO,GP1        ;   turn off LED

        goto    loop            ; repeat forever
```

Note that the logic seems to be inverse; the LED is turned on if GP3 is clear, yet the '`btfss`' instruction tests for the GP3 bit being set. Since the bit test instructions skip the next instruction if the bit test condition is met, the instruction following a bit test is executed only if the condition is *not* met. Often, following a bit

test instruction, you'll place a '`goto`' or '`call`' to jump to a block of code that is to be executed if the bit test condition is not met. In this case, there is no need, as the LED can be turned on or off with single bit set or clear instructions.

However, as discussed in lesson 1, directly setting or clearing individual bits in an I/O port can lead to unintended effects, due a potential read-modify-write problem – you may find that bits other than the designated one are also being changed. This unwanted effect often occurs when sequential bit set/clear instructions are performed on the same port. Trouble can be avoided by separating sequential '`bsf`' and '`bcf`' instructions with a '`nop`'.

Although unlikely to be necessary in this case, since the bit set/clear instructions are not sequential, a shadow register could be used as follows:

```
        ; initialisation
        movlw   ~(1<<GP1)       ; configure GP1 (only) as an output
        banksel TRISIO          ; (GP3 is an input)
        movwf   TRISIO

        banksel GPIO
        clrf    GPIO            ; start with GPIO clear (LED off)
        clrf    sGPIO           ; update shadow copy

;***** Main loop
loop
        btfss   GPIO,GP3        ; if button pressed (GP3 low)
        bsf     sGPIO,GP1       ;   turn on LED
        btfsc   GPIO,GP3        ; if button up (GP3 high)
        bcf     sGPIO,GP1       ;  turn off LED

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        goto    loop            ; repeat forever
```

It's possible to optimise this a little. There is no need to test for button up as well as button down; it will be either one or the other, so we can instead write a value to the shadow register, assuming the button is up (or down), and then test just once, updating the shadow if the button is found to be down (or up).

The main loop then becomes:

```
loop
        clrf    sGPIO           ; assume button up -> LED off
        btfss   GPIO,GP3        ; if button pressed (GP3 low)
        bsf     sGPIO,GP1       ;   turn on LED

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        goto    loop            ; repeat forever
```

It's also not really necessary to initialise GPIO at the start; whatever state it is in when the program starts, it will be updated the first time the loop completes, a few µs later – much too fast to see. If setting the initial values of output pins correctly is important, to avoid power-on glitches that may affect circuits connected to them, the correct values should be written to the port registers before configuring the pins as outputs, i.e. initialise GPIO before TRISIO. But when dealing with human perception, it's not important.

If you didn't use a shadow register, but tried to take the same approach – assuming a state (e.g. "button up"), setting GPIO, then reading the button and changing GPIO accordingly – it would mean that the LED would be flickering on and off, albeit too fast to see. Using a shadow register is a neat solution that avoids this

problem, as well as any read-modify-write concerns, since the physical register (GPIO) is only ever updated with the correctly determined value.

Complete program

Here is the complete program for turning on the LED when the pushbutton is pressed, using the optimised shadow register code above:

```
;*****************************************************************
;   Demonstrates use of shadow register when reading a switch           *
;                                                                       *
;   Turns on LED when pushbutton on GP3 is pressed (active low)         *
;*****************************************************************
;   Pin assignments:                                                    *
;       GP1 - LED                                                       *
;       GP3 - pushbutton switch                                         *
;*****************************************************************

    list        p=12F629
    #include    <p12F629.inc>

    errorlevel  -302            ; no warnings about registers not in bank 0


;***** CONFIGURATION
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT


;***** VARIABLE DEFINITIONS
        UDATA_SHR
sGPIO   res 1                   ; shadow copy of GPIO


;*****************************************************************
RESET   CODE    0x0000          ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   then update OSCCAL
        movwf   OSCCAL

;***** Main program
        ; initialisation
        movlw   ~(1<<GP1)       ; configure GP1 (only) as an output
        banksel TRISIO          ; (GP3 is an input)
        movwf   TRISIO

;***** Main loop
        banksel GPIO            ; select bank for GPIO access
loop
        clrf    sGPIO           ; assume button up -> LED off
        btfss   GPIO,GP3        ; if button pressed (GP3 low)
        bsf     sGPIO,GP1       ;   turn on LED

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        goto    loop            ; repeat forever

        END
```

Debouncing

In most applications, you want your code to respond to transitions; some action should be triggered when a button is pressed or a switch is toggled. This presents a problem when interacting with real, physical switches, because their contacts *bounce*. When most switches change, the contacts in the switch will make and break a number of times before settling into the new position. This contact bounce is generally too fast for the human eye to see, but microcontrollers are fast enough to react to each of these rapid, unwanted transitions.

A similar problem can be caused by *electromagnetic interference (EMI)*. Unwanted spikes may appear on an input line, due to electromagnetic noise, especially (but not only) when switches or sensors are some distance from the microcontroller. But any solution which deals effectively with contact bounce will generally also remove or ignore input spikes caused by EMI.

Dealing with these problems is called switch *debouncing*.

To illustrate the problem, suppose that you wish to toggle the LED on GP1, once, each time the button on GP3 is pressed.

In pseudo-code, this could be expressed as:

```
do forever
      wait for button press
      toggle LED
      wait for button release
end
```

Note that it is necessary to wait for the button to be released before restarting the loop, so that the LED should only toggle once per button press. If we didn't wait for the button to be released before continuing, the LED would continue to toggle as long as the button was held down; not the desired behaviour.

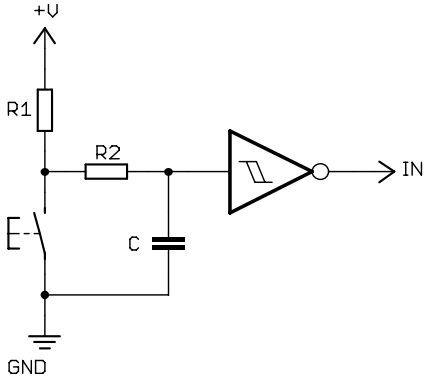
Here is some code which implements this:

```
;***** Main program
        ; initialisation
        movlw   ~(1<<GP1)       ; configure GP1 (only) as an output
        banksel TRISIO          ; (GP3 is an input)
        movwf   TRISIO
        banksel GPIO
        clrf    GPIO            ; start with LED off
        clrf    sGPIO           ;   update shadow

;***** Main loop
loop

waitdn  btfsc   GPIO,GP3        ; wait until button pressed (GP3 low)
        goto    waitdn

        movf    sGPIO,w
        xorlw   1<<GP1          ; toggle LED on GP1
        movwf   sGPIO           ;   using shadow register
        movwf   GPIO

waitup  btfss   GPIO,GP3        ; wait until button released (GP3 high)
        goto    waitup          ;   before continuing

        goto    loop            ; repeat forever
```

If you build this program and test it, you will find that it is difficult to reliably change the LED when you press the button; sometimes it will change, other times not. This is due to contact bounce.

In baseline lesson 4 we saw that switch debouncing is in effect a filtering problem and that it can be addressed by using appropriate hardware.

One solution is to use an RC low-pass filter coupled to a Schmitt trigger buffer, as shown on the right.



However, one of the reasons to use microcontrollers is that they allow you to solve what would otherwise be a hardware problem, in software. In particular, it is possible to use software routines to debounce a switch input, without any need for external filtering hardware.

If the software can ignore input transitions due to contact bounce or EMI, while detecting and responding to genuine switch changes, no external debounce circuitry is needed. As with the hardware approach, the problem is essentially one of filtering; we need to ignore any transitions too short to be 'real'.

Example 2: Debouncing using Delays

The easiest approach to software debouncing is to estimate the maximum time the switch could possibly take to settle, and then simply wait at least that long, after detecting the first transition. If the wait time, or delay, is longer than the maximum possible settling time, you can be sure that, by the time the delay completes, the switch will have finished bouncing.

It's simply a matter of adding a suitable debounce delay, after each transition is detected, as in the following pseudo-code:

```
do forever
      wait for button press
      toggle LED
      delay debounce_time
      wait for button release
      delay debounce_time
end
```

Note that the LED is toggled immediately after the button press is detected. There's no need to wait for debouncing. By acting on the button press as soon as it is detected, the user will experience as fast a response as possible.

The necessary minimum delay time depends on the characteristics of the switch. For example, the switch tested in baseline lesson 4 was seen to settle in around 250 µs. Repeated testing showed no settling time greater than 1 ms, but it's difficult to be sure of that, and perhaps a different switch, say that used in production hardware, rather than the prototype, may behave differently. So it's best to err on the safe side, and choose the longest delay we can get away with. People don't notice delays of 20 ms or less (flicker is only barely perceptible at 50Hz, corresponding to a 20 ms delay), so a good choice is probably 20 ms.

As you can see, choosing a suitable debounce delay is not an exact science!

To generate a 20 ms delay, we can use the W × 10 ms delay module developed in lesson 2.

It is then straightforward to add delays to the main loop of the "Toggle an LED" code (presented above), as follows:

```
loop
        banksel GPIO            ; wait until button pressed (GP3 low)
waitdn  btfsc   GPIO,GP3
        goto    waitdn

        movf    sGPIO,w
        xorlw   1<<GP1          ; toggle LED on GP1
        movwf   sGPIO           ;   using shadow register
        movwf   GPIO

        movlw   .2              ; delay 20 ms to debounce (GP3 low)
        pagesel delay10
        call    delay10
        pagesel $

        banksel GPIO            ; wait until button released (GP3 high)
waitup  btfss   GPIO,GP3
        goto    waitup          ;   before continuing

        movlw   .2              ; delay 20 ms to debounce (GP3 high)
        pagesel delay10
        call    delay10
        pagesel $

        goto    loop            ; repeat forever
```

Note the extra '`banksel`' directives; these have been added in case the '`delay10`' routine changes the current bank selection. That's not strictly necessary in this case, because we know that this version of the '`delay10`' routine does not affect the current bank selection (it only uses shared registers). But in general it's safer to assume that, when you call a subroutine, it may change both the bank and page selection bits (hence the '`pagesel $`' directive following each call to the delay routine – it ensures that the subsequent '`goto`' instructions in the routine will work correctly).

If you build and test this code, you should find that the LED now reliably changes state every time you press the button.

Example 3: Debouncing using a Counting Algorithm

There are a couple of problems with using a fixed length delay for debouncing.

Firstly, the need to be "better safe than sorry" means making the delay as long as possible, and probably slowing the response to switch changes more than is really necessary, potentially affecting the feel of the device you're designing.

More importantly, the delay approach cannot differentiate between a glitch and the start of a switch change. As discussed, spurious transitions can be caused be EMI, or electrical noise – or a momentary change in pressure while a button is held down.

A commonly used approach, which avoids these problems, is to regularly read (or *sample*) the input, and only accept that the switch is in a new state, when the input has remained in that state for some number of times in a row. If the new state isn't maintained for enough consecutive times, it's considered to be a glitch or a bounce, and is ignored.

For example, you could sample the input every 1 ms, and only accept a new state if it is seen 10 times in a row; i.e. high or low for a continuous 10 ms.

To do this, set a counter to zero when the first transition is seen. Then, for each sample period (say every 1 ms), check to see if the input is still in the desired state and, if it is, increment the counter before checking

again. If the input has changed state, that means the switch is still bouncing (or there was a glitch), so the counter is set back to zero and the process restarts. The process finishes when the final count is reached, indicating that the switch has settled into the new state.

The algorithm can be expressed in pseudo-code as:

```
count = 0
while count < max_samples
      delay sample_time
      if input = required_state
            count = count + 1
      else
            count = 0
end
```

Here is the modified "toggle an LED" main loop, illustrating the use of this counting debounce algorithm:

```
loop

db_dn   ; wait until button pressed (GP3 low), debounce by counting:
        clrf    db_cnt
        clrf    dc1
dn_dly  incfsz  dc1,f           ;   delay 256x3 = 768us.
        goto    dn_dly
        btfsc   GPIO,GP3        ;   if button up (GP3 set),
        goto    db_dn           ;       restart count
        incf    db_cnt,f        ;   else increment count
        movlw   .13             ;   max count = 10ms/768us = 13
        xorwf   db_cnt,w        ;   repeat until max count reached
        btfss   STATUS,Z
        goto    dn_dly

        ; toggle LED on GP1
        movf    sGPIO,w
        xorlw   1<<GP1          ; toggle LED on GP1
        movwf   sGPIO           ;   using shadow register
        movwf   GPIO

db_up   ; wait until button released (GP3 high), debounce by counting:
        clrf    db_cnt
        clrf    dc1
up_dly  incfsz  dc1,f           ;   delay 256x3 = 768us.
        goto    up_dly
        btfss   GPIO,GP3        ;   if button down (GP3 clear),
        goto    db_up           ;       restart count
        incf    db_cnt,f        ;   else increment count
        movlw   .13             ;   max count = 10ms/768us = 13
        xorwf   db_cnt,w        ;   repeat until max count reached
        btfss   STATUS,Z
        goto    up_dly

        ; repeat forever
        goto    loop
```

There are two debounce routines here; one for the button press, the other for button release. The program first waits for a pushbutton press, debounces the press, then toggles the LED before waiting for the pushbutton to be released, and then debouncing the release.

The only difference between the two debounce routines is the input test: '`btfsc GPIO,3`' when testing for button up, versus '`btfss GPIO,3`' to test for button down.

Note that, in each of the debounce routines, a short loop is used to generate a 768 µs delay, so the input is being sampled every 768 µs or so, instead of the 1 ms sample time mentioned above – simply because it's much easier to generate a 768 µs delay than a 1 ms delay. The principle is the same; instead of sampling the input 10 times, 1 ms apart, the routine samples 13 times, 768 µs apart. Either way, the routine is checking that the switch is remaining in the same state (continuously on or off) for approximately 10 ms.

The code above demonstrates one method for counting up to a given value (13 in this case):

The count is zeroed at the start of the routine.

It is incremented within the loop, using the '`incf`' instruction – "**inc**rement **f**ile register". As with many other instructions, the incremented result can be written back to the register, by specifying '`,f`' as the destination, or to W, by specifying '`,w`' – but normally you would use it as shown, with '`,f`', so that the count in the register is incremented. The midrange PICs also provide a '`decf`' instruction – "**dec**rement **f**ile register", which is similar to '`incf`', except that it performs a decrement instead of increment.

We've seen the '`xorwf`' instruction before, but not used in quite this way. The result of exclusive-oring any binary number with itself is zero. If any dissimilar binary numbers are exclusive-ored, the result will be non-zero. Thus, XOR can be used to test for equality, which is how it is being used here. First, the maximum count value is loaded into W, and then this max count value in W is xor'd with the loop count. If the loop counter has reached the maximum value, the result of the XOR will be zero. We do not care what the result of the XOR actually is, only whether it is zero or not. And to avoid overwriting the loop counter with the result, '`,w`' is specified as the destination of the '`xorwf`' instruction – writing the result to W, effectively discarding it.

To check whether the result of the XOR was zero (which will be true if the count has reached the maximum value), we use the '`btfss`' instruction to test the zero flag bit, Z, in the STATUS register.

Alternatively, each debounce routine could have been coded by initialising the loop counter to the maximum value at the start of the loop, and using '`decfsz`' at the end of the loop, as follows:

```
db_dn   ; wait until button pressed (GP3 low), debounce by counting:
        movlw   .13             ; max count = 10ms/768us = 13
        movwf   db_cnt
        clrf    dc1
dn_dly  incfsz  dc1,f           ; delay 256x3 = 768us.
        goto    dn_dly
        btfsc   GPIO,GP3        ; if button up (GP3 set),
        goto    db_dn           ;   restart count
        decfsz  db_cnt,f        ; else repeat until max count reached
        goto    dn_dly
```

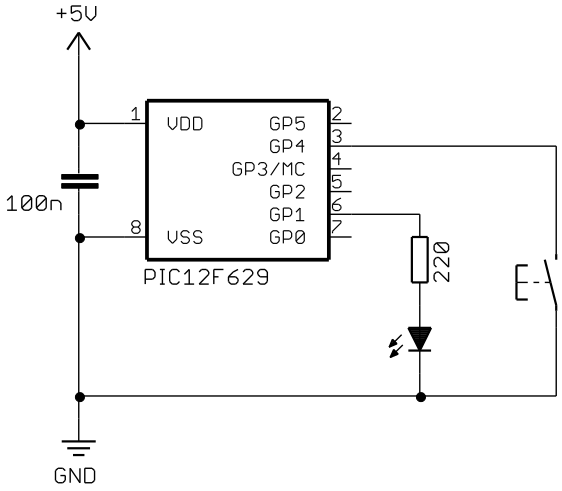
That's two instructions shorter, and at least as clear, so it's a better way to code this routine.

Nevertheless it's worth knowing how to count up to a given value, using XOR to test for equality, as shown above, because sometimes it simply makes more sense to count up than down.

Example 4: Internal Pull-ups

The use of pull-up resistors is so common that most modern PICs make them available internally, on at least some of the pins.

The availability of internal pull-ups makes it possible to do away with the external pull-up resistor, as shown in the circuit on the right.



Unfortunately, there is no internal pull-up on the 12F629's GP3 pin, so to demonstrate their use we need to use a different input pin, which is why the switch is connected to GP4 in this circuit.

Strictly speaking, the internal pull-ups are not simple resistors. Microchip refers to them as "weak pull-ups"; they provide a small current (typically 250 µA) which is enough to hold a disconnected, or *floating*, input high, but not enough to strongly resist any external signal trying to drive the input low.

We've seen that, on baseline PICs, internal pull-ups are only available on a few pins, and they are either all enabled or all disabled.

This is different in the midrange architecture: pull-ups are available for every pin on the 12F629 (except GP3), and they can be selected individually.

Nevertheless, the internal weak pull-ups are globally controlled, as a group, by the $\overline{\text{GPPU}}$ bit in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION_REG	$\overline{\text{GPPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

By default (after a power-on or reset), $\overline{\text{GPPU}}$ = 1 and all the internal pull-ups are disabled.

To globally enable internal pull-ups, clear $\overline{\text{GPPU}}$.

Note that, in the midrange architecture, the OPTION register is accessed as a normal, memory-mapped register, called OPTION_REG, as mentioned in lesson 1.

> *Note: The `option` instruction is **not** used to write to the OPTION register on midrange devices. The OPTION register is accessed as OPTION_REG, using general instructions, such as `bsf`.*

Each weak pull-up is individually controlled by a bit in the WPU register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
WPU	-	-	WPU5	WPU4	-	WPU2	WPU1	WPU0

If WPU<n> = 1, the weak pull-up on the corresponding GPIO pin, GPn, is enabled.

If WPU<n> = 0, the corresponding weak pull-up is disabled.

However, if a pin is configured as an output, the internal pull-up is automatically disabled for that pin.

To enable the pull-up on GP4, we must first clear $\overline{\text{GPPU}}$ to globally enable weak pull-ups:

```
bcf     OPTION_REG,NOT_GPPU     ; enable global pull-ups
```

One advantage of the midrange architecture is that, because the OPTION register can be accessed directly, it is possible to clear or set individual bits, such as $\overline{\text{GPPU}}$, leaving the other bits unchanged. On the baseline PICs, we had to load the whole OPTION register in a single operation, which is much less convenient.

Then, having globally enabled weak pull-ups, we need to specifically enable the pull-up on GP4, by setting WPU<4>.

You could do that by:

```
bsf     WPU,GP4         ; enable pull-up on GP4
```

By default (after a power-on reset), every bit of WPU is set, so there is not really any need to explicitly set WPU<4> like this. But it's good practice to disable the weak pull-ups on the unused input pins (unused inputs should not be left floating, to avoid large current consumption and ESD damage to the PIC, and are often tied to ground; if pull-ups were enabled on grounded inputs, current will flow through them, leading to unnecessary power consumption). So all the remaining bits in WPU should be cleared.

This could be done by:

```
clrf    WPU             ; disable all pull-ups
bsf     WPU,GP4         ; except on GP4
```

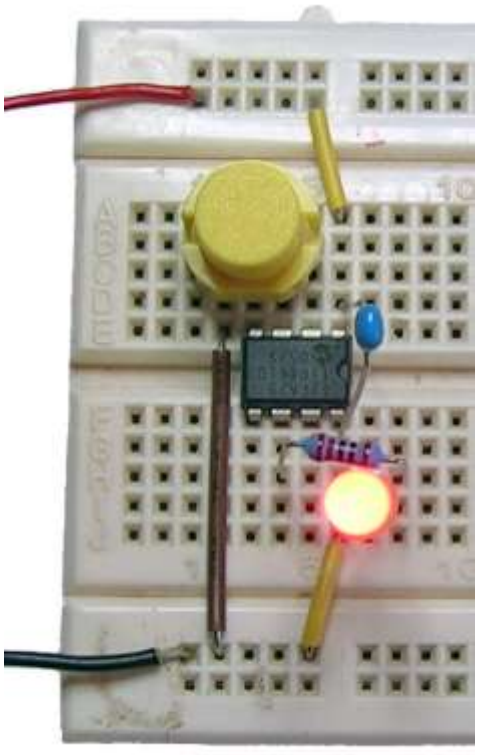
or:

```
movlw   1<<GP4          ; select pull-up on GP4 only
movwf   WPU
```

The second form is better if you need to enable pull-ups on more than one input.

To build this circuit, you could connect a pushbutton between GP4 (available on pin 2 of the 14-pin header on the Low Pin Count Demo Board) and ground (pin 14 on the header).

Or, if you wish to really demonstrate to yourself that there is definitely no external pull-up resistor connected to the GP4 input, you can remove the PIC from the LPC Demo Board (after programming it!), and place it in your own circuit, which you could build using prototyping breadboard, as illustrated on the right.



Note that this minimal circuit, diagrammed above and illustrated here, does not include a current-limiting resistor between GP4 and the pushbutton. As discussed earlier, that's generally ok, but to be safe, it's good practice to include a current limiting resistor, of around 1 kΩ, between the PIC pin and the pushbutton.

But as this example illustrates, functional PIC-based circuits really can be built with very few external components!

Complete program

Here's the complete "Toggle an LED" program, illustrating how to read and debounce a simple switch on a pin held high by an internal pull-up:

```
;*****************************************************************
;                                                                *
;   Description:    Lesson 1, example 4                          *
;                                                                *
;   Demonstrates use of internal pull-ups plus debouncing        *
;                                                                *
;   Toggles LED when pushbutton is pressed (low) then released (high)   *
;   Uses counting algorithm to debounce switch                   *
;                                                                *
;*****************************************************************
;                                                                *
;   Pin assignments:                                             *
;       GP1 - LED                                                *
;       GP4 - pushbutton switch                                  *
;                                                                *
;*****************************************************************

    list        p=12F629
    #include    <p12F629.inc>

    errorlevel  -302            ; no warnings about registers not in bank 0


;***** CONFIGURATION
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT


;***** VARIABLE DEFINITIONS
        UDATA_SHR
sGPIO   res 1                   ; shadow copy of GPIO
db_cnt  res 1                   ; debounce counter
dc1     res 1                   ; delay counter


;*****************************************************************
RESET   CODE    0x0000          ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   then update OSCCAL
        movwf   OSCCAL

;***** Initialisation
        ; configure port
        movlw   ~(1<<GP1)       ; configure GP1 (only) as an output
        banksel TRISIO          ; (GP4 is an input)
        movwf   TRISIO
        ; enable weak pull-up on switch input
        banksel OPTION_REG      ; enable global pull-ups
        bcf     OPTION_REG,NOT_GPPU
        movlw   1<<GP4          ; select pull-up on GP4 only
        banksel WPU
        movwf   WPU
        ; initialise port
        banksel GPIO            ; start with LED off
```

```
        clrf    GPIO
        clrf    sGPIO           ;   update shadow

;***** Main loop
loop

db_dn   ; wait until button pressed (GP4 low), debounce by counting:
        movlw   .13             ; max count = 10ms/768us = 13
        movwf   db_cnt
        clrf    dc1
dn_dly  incfsz  dc1,f           ; delay 256x3 = 768us.
        goto    dn_dly
        btfsc   GPIO,GP4        ; if button up (GP4 set),
        goto    db_dn           ;   restart count
        decfsz  db_cnt,f        ; else repeat until max count reached
        goto    dn_dly

        ; toggle LED on GP1
        movf    sGPIO,w
        xorlw   1<<GP1          ; toggle LED on GP1
        movwf   sGPIO           ;   using shadow register
        movwf   GPIO

db_up   ; wait until button released (GP4 high), debounce by counting:
        movlw   .13             ; max count = 10ms/768us = 13
        movwf   db_cnt
        clrf    dc1
up_dly  incfsz  dc1,f           ; delay 256x3 = 768us.
        goto    up_dly
        btfss   GPIO,GP4        ; if button down (GP4 clear),
        goto    db_up           ;   restart count
        decfsz  db_cnt,f        ; else repeat until max count reached
        goto    up_dly

        ; repeat forever
        goto    loop


        END
```

That's enough on reading switches for now. There's plenty more to explore, of course, such as reading keypads and debouncing multiple switch inputs – topics to explore later.

But in the next lesson we'll look at the PIC12F629's 8-bit timer module, Timer0.